You should work on the following assignments in fixed teams of two. Please note that *every* team member must be able to explain *all* solutions of the team of two. Please submit only one solution for each team of two. You can implement the assignments within the same programming project, i.e., you do not have to create a .c-file for every assignment. Please take care of the programming guideline (can be found in our moodle room) and handling of errors and special cases!

**Deadline to upload your solution for at least assignments 1, 2, 3, and 4.a:**
**Three days bevor the laboratory**
Assignments 4.b and 5 can be done during the laboratory.

If you have questions or if you need any help, use the forum in our moodle room und help each other.

*Assignment 1: Constructing the Array*
Define a macro for the maximum array size `MAX_SIZE` of 20. In function `main`, define an integer array of maximum array size. The integer array should be initialized using the function `initArray`. Implement function `initArray` with two parameters:

- one for the integer array to be initialized
- one for an integer value `k`.

The function should work as follows:

- The array shall be initialized with 1 to `MAX_SIZE` in ascending order.
- Afterwards the function shall pick k many elements randomly and put them at the end of the array. Hence, the array should have `MAX_SIZE-k` sorted elements at the beginning and k many unsorted elements at the end. Example for k=5:

| 1 | 2 | 3 | 6 | 7 | 8 | 9 | 10 | 11 | 13 | 14 | 16 | 17 | 19 | 20 | 4 | 15 | 12 | 5 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- The function shall handle the case k>`MAX_SIZE` as the case `MAX_SIZE`=k.

*Assignment 2: Implementing sorting algorithms introduced in the lecture*
Implement the following sorting algorithms as given in the lectures which get the array to be sorted as parameter.

a) InsertionSort
b) SelectionSort
c) ShellSort
   (gap interval is defined as $\frac{MAX\_SIZE}{2^t}$; e.g., if MAX_SIZE = 13, then the gaps are 6, 3, 1)
d) MergeSort
e) Bottom-up MergeSort
f) QuickSort

*Assignment 3: Counter*

Extend the implementations of the Assignment 2 by counters for the number of both exchanges and comparisons. Create a table with the results that can be used to compare the different sorting algorithms.

NOTE: To simplify the implementation of the counters, you may exceptionally use global variables for the counters in this assignment.

*Assignment 4: Evaluation*

Evaluate the performance of the sorting algorithms given in Assignment 2 by

a) constructing arrays with the help of function `initArray` of Assignment 1 with growing k. Measure the computation times needed. For these experiments fix an appropriate maximum array size (e.g., $50.000$). Use SelectionSort and MergeSort as sorting algorithms. In each iteration double the value of k for initializing the array until k ≥ `MAX_SIZE` holds. If the run of a sorting algorithm exceeds a computation time of 5 minutes you can stop the run for this one.
   Examine whether (and how) the computation times grows with k. Moreover, examine the numbers of comparisons.
b) constructing arrays with the help of the function `initArray` of Assignment 1 with growing `MAX_SIZE`=k. Measure the computation times needed. For these experiments run all sorting algorithms with
   a. `MAX_SIZE=k=100`
   b. `MAX_SIZE=k=200`
   c. `MAX_SIZE=k=500`
   d. `MAX_SIZE=k=1.000`

   If an algorithm exceeds a computation time of 5 minutes you can stop the run for this one. Examine whether (and how) the computation times grows with `MAX_SIZE`=k. Moreover, examine the numbers of exchanges and comparisons.

Notice that all computation time measures of (a) and (b) should be computed as average of at least 5 runs.

*Assignment 5:*

Implement the following two sorting algorithms and counter the number of both their exchanges and their comparisons, so that you can compare them to the sorting algorithms of Assignment 2.

a) BubbleSort

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

See also https://en.wikipedia.org/wiki/Bubble_sort

b) CombSort:

```
function combsort(array input) is

    gap := input.size // Initialize gap size
    shrink := 1.3 // Set the gap shrink factor
    sorted := false

    loop while sorted = false
        // Update the gap value for a next comb
        gap := floor(gap / shrink)
        if gap ≤ 1 then
            gap := 1
            sorted := true // If there are no swaps this pass, we are done
        end if

        // A single "comb" over the input list
        i := 0
        loop while i + gap < input.size // See Shell sort for a similar idea
            if input[i] > input[i+gap] then
                swap(input[i], input[i+gap])
                sorted := false
                // If this assignment never happens within the loop,
                // then there have been no swaps and the list is sorted.
             end if

             i := i + 1
         end loop
     end loop
end function
```

See also https://en.wikipedia.org/wiki/Comb_sort

Note: The floor function is the function that takes as input a real number x and gives as output the greatest integer less than or equal to x, i.e., round down.